Micro Digital

smxNOR™ User's Guide

NOR Flash Driver

Version 1.12
September 14, 2012

by Yingbo Hu

Table of Contents

1. Overview **1**

2. Using smxNOR **1**
2.1 Installation 1
2.2 Getting Started 1
2.3 Basic Terms 1
2.4 Configuration Settings 1

3. Theory of Operation **4**

4. NOR Flash Driver API **5**
4.1 API Data Types 5
4.2 API Reference 5

5. NOR Flash Hardware IO Routines **10**
5.1 IO Routine Data Types 10
5.2 IO Routine Reference 10
5.3 Verify the Driver 14
5.4 Erase Your Flash First 15

6. Application Notes **16**
6.1 Support Multiple Flash Chips as One Big Disk 16
6.2 Support Multiple Flash Chips as Multiple Disks 16
6.3 Porting for CFI-Compatible Flash 19

A. File Summary **20**

B. Porting Notes **21**
B.1 C Library Function Requirements 21

C. Tested Hardware **22**

D. Preprogramming Flash **23**

© Copyright 2006-2012

Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

All rights reserved.

smxNOR is a Trademark of Micro Digital Inc.
smx is a Registered Trademark of Micro Digital Inc.

1. Overview

smxNOR is a NOR flash driver designed for embedded systems to have a minimal RAM requirement. Typically 700 bytes is sufficient to achieve moderate performance. It supports Common Flash Interface (CFI) NOR chips and some others.

The following restrictions must be applied to make it work:

- Assume the NOR driver uses only regular size blocks. Small blocks will be left for other uses, such as boot, program storage, etc. If smaller blocks are half the size of regular blocks, you can just group two together and treat them as a regular block.
- Supported NOR flash types are those that: support block erase (change all cells to 0xFF) and each byte or word can be written at least 3 times (but bits can be changed only from ‘1’ to ‘0’).
- The Sector Map requires space, in bytes, equal to four times the number of sectors per block. Normally this will fit into the first sector of each block. Any space left over is not used.

If you have any question about the above restrictions, please contact us.

2. Using smxNOR

2.1 Installation

smxNOR is installed by copying files from the distribution media. When ordered with the smxFS, it is part of the smxFS release and is installed with it.

2.2 Getting Started

smxNOR is configured to support smxFS and the processors and compilers it supports. If you are using smxNOR for another file system, processor, or compiler, see section B Porting Notes, and implement the porting layer for your environment first, before using smxNOR.

2.3 Basic Terms

Block Minimum erasable unit of the flash chip. Some flash chips may use the term *sector*, instead.
Sector Minimum unit of data for a file system. It is 512 bytes, by default.
Page Maximum reading/writing unit of data of the flash chip. It is 256 or 512 bytes.

2.4 Configuration Settings

If you change any settings you should rebuild the smxNOR library, clean.

2.4.1 fdcfg.h

fdcfg.h contains flash driver configuration constants that allow you select features and tune performance, code size, and RAM usage.

NOR_MAX_CHIP_NUM

This specifies how many physical NOR flash chips are in your system. Default value is 1.

NOR_PSMC_ENTRY_NUM

This is the internal Flash Driver mapping table cache size. Increasing it will increase the RAM requirement. Each mapping table item will use two bytes or four bytes, depending on the flash size, so if you have enough memory, increase it to 64 or 128. The default setting is 16.

NOR_PSMC_ITEM_NUM

This is the internal Flash Driver mapping table cache number. Increasing it will increase the RAM requirement. It must be greater than 1.

NOR_DATA_READ_BACK_CHECK

Set it to "1" to read back the data after writing to verify it. Read back will increase the reliability but decrease the performance. We recommend enabling it only when you are debugging your code.

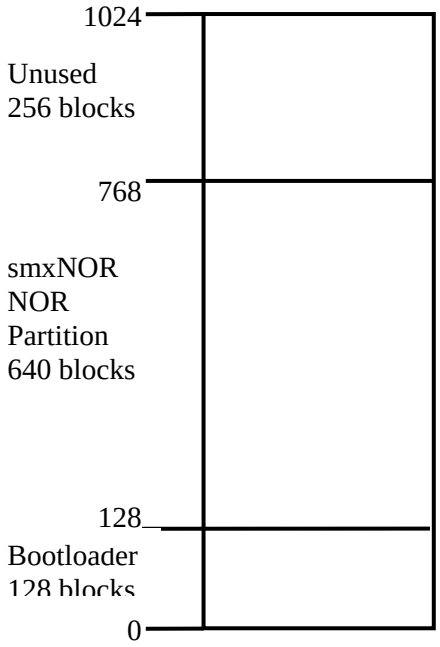
NOR_WEAR_LEVELING_THRESHOLD

This is the threshold for static wear leveling. Since v1.12, static wear leveling is supported. We save the wear counter of each block so if the maximum wear counter is greater than the least wear counter plus this threshold, the function nor_WearLeveling will move the data in the block which has the least wear counter to other place so this block can be freed and used by a later write operation.

NOR_START_BLOCK_INDEX
NOR_ BLOCK_NUM

These specify the start and size of the partition used by smxNOR. By default NOR_START_BLOCK_INDEX is 0 and NOR_BLOCK_NUM is –1, which means it will use the whole flash memory area. The following example reserves 128 blocks for a bootloader and the partition is 640 blocks, leaving 256 blocks above for some other purpose.

```
#define NOR_START_BLOCK_INDEX       128
#define NOR_BLOCK_NUM                640
```



2.4.2 fdport.h

fdport.h contains OS and compiler definitions. smxNOR's porting layer maps onto smxBase services. Please see the smxBase User's Guide for detailed information.

3. Theory of Operation

smxNOR maps the logical sector index passed to the functions nor_SectorRead(), nor_SectorWrite(), and nor_SectorDiscard() to the physical address within the NOR flash memory. The mapping table is stored at the beginning of each flash block and contains the logical sector index and status information of the sector for each physical sector within that block. The way the map is stored in flash is not convenient for normal operation, so a sector map cache is maintained in RAM to improve performance. An algorithm has been chosen to achieve the best performance under normal conditions. The cache size is determined at compile time and can be adjusted depending upon the NOR flash partition size.

When smxNOR needs to update the contents of a sector, it will never write the data to the same place as the old data; it will first select a new empty sector, mark it as in-progress, write the updated data to the new place, mark the data as valid, and then mark the old data sector as discarded. So whenever the power fails during this write operation, the old data will be untouched. That is the reason why smxNOR is power fail safe at the sector level. On power up the entire flash partition is scanned for a sector marked in-progress. If found then the flash partition is scanned to find the old sector having the same logical address. If the old sector status is valid, then the new sector is deemed to be only partially written, and it is discarded. But if the old sector status is discarded, then the new sector status is changed to valid.

When the number of empty sectors in the flash drops below a specified threshold, smxNOR automatically collects the discarded old sectors. It moves the valid sectors within a block to other blocks and then erases the whole discarded block. During this time, the NOR flash driver is busy and will not accept read or write operations. To minimize downtime, only one flash block is reclaimed, at a time.

Static wear leveling is done by the API function nor_WearLeveling(), which uses wear counters and a configurable threshold. Dynamic wear leveling is a natural result of always moving pointers through flash memory in an increasing direction and recycling from the bottom when the top of the flash partition reached.

4. NOR Flash Driver API

The smxNOR flash driver API is defined in norfd.h, which contains the functions that are called by the file system. These comprise the high-level driver.

int **nor_FlashInit** (uint iID);

int **nor_FlashRelease** (uint iID);

int **nor_BlockReclaim** (uint iID, u32 iExpectedEmptySectorNum);

int **nor_SectorRead** (uint iID, u8 * pRAMAddr, u32 wLogicalIndex);

int **nor_SectorWrite** (uint iID, u8 * pRAMAddr, u32 wLogicalIndex);

int **nor_SectorDiscard**(uint iID, u32 wLogicalIndex);

u32 **nor_SectorNum** (uint iID);

u32 **nor_SectorSize** (uint iID);

int **nor_WearLeveling** (uint iID);

4.1 API Data Types

These are defined in **fdport.h**.

u32, u16, etc Unsigned integer types of the size (bits) indicated.

4.2 API Reference

int **nor_FlashInit** (uint iID)

Summary Initialize smxNOR flash driver.

Details This function should be called first before you use the NOR flash driver. It calls the NOR flash hardware IO routine to initialize the NOR flash chip and retrieve the basic information of the NOR flash chip such as the block size and total block number. It then tries to read the flash chip to find if this flash chip was used before by smxNOR flash driver. If so, it retrieves the old information so the file system will get the saved data; otherwise, this function will format it.

Please make sure the flash chip is empty, meaning all the data bytes are FF before you first use the chip. The driver will NOT automatically erase the whole chip before using it.

Pars nID The device ID you want to use. Valid values are 0 to NOR_MAX_CHIP_NUM-1.

Returns 1 Initialization succeeded.
0 Initialization failed.

See Also nor_FlashRelease()

Example

```
If(nor_FlashInit(0))
    printf("NOR Flash Initialized.");
```

int **nor_FlashRelease** (uint iID)

Summary Release smxNOR flash driver.

Details This function should be called when you are done with the flash driver. It releases the internal buffers allocated by nor_FlashInit() and calls the NOR flash hardware IO routine to release the hardware resource.

Pars nID The device ID you want to use (the same ID you passed to nor_FlashInit()).

Returns 1 Release succeeded.
0 Release failed.

See Also nor_FlashInit()

Example

```
If(nor_FlashRelease(0))
    printf("NOR Flash Released.");
```

int **nor_BlockReclaim** (uint iID, u32 iExpectedEmptySectorNum)

Summary Do block reclaim for smxNOR flash driver.

Details This function will reclaim the used blocks to get more empty sectors that can be used by the file system.The driver will be forced to do Block Reclaim when the number of empty sectors has dropped to certain low level, but we recommend you call this API when your system is idle to improve performance when writing data. smxFS already provides an IOCTL function to do it. If you are using smxFS, please call that function to make sure it is multitasking safe.

Pars

nID	The device ID you want to use (the same ID you passed to nor_FlashInit()).
iExpectedEmptySectorNum	The expected number of empty sectors you want to get after the block Reclaim returns.

Returns The current number of empty sectors on the NOR flash chip.

See Also nor_FlashInit()

Example

```
printf("Current Empty Sector Number is %d\n.", nor_BlockReclaim(0, 0));
```

int **nor_SectorRead** (uint iID, u8 * pRAMAddr, u32 wLogicalIndex)

Summary Read one sector of data from the flash chip.

Details This function reads one sector of data from the flash chip. A sector is normally 512 bytes by default but may be another value so please make sure the memory buffer is big enough.

Pars

nID	The device ID you want to use (the same ID you passed to nor_FlashInit()).
pRAMAddr	The pointer to the memory buffer to hold the data. It should be at least one sector in size.
wLogicalIndex	The logical sector index from the file system's point of view.

Returns 0 The read succeeded.

See Also nor_SectorWrite()

Example

```
u8 pData[512];
nor_SectorRead(0, pData, 0); /* Read the MBR if using FAT file system */
```

int **nor_SectorWrite** (uint iID, u8 * pRAMAddr, u32 wLogicalIndex)

Summary Write one sector of data to the flash chip.

Details This function writes one sector of data to the flash chip. A sector is normally 512 bytes by default.

Pars

nID	The device ID you want to use (the same ID you passed to nor_FlashInit()).
pRAMAddr	The pointer to the memory buffer holding the data. It should be at least one sector in size.
wLogicalIndex	The logical sector index from the file system's point of view.

Returns 0 The write succeeded.

See Also nor_SectorRead()

Example

```
u8 pData[512];
memset(pData, 0xFF, 512);
nor_SectorWrite(0, pData, 0); /* Empty the first sector of file system */
```

int **nor_SectorDiscard** (uint iID, u32 wLogicalIndex)

Summary Tell theflash driver that a sector is not used by the file system.

Details This function will mark one sector as discarded, which means the file system does not use the data in this sector so the driver can reclaim it later when it does block reclaim.

Pars

nID	The device ID you want to use (the same ID you passed to nor_FlashInit()).
wLogicalIndex	The logical sector index of the discarded sector.

Returns 0

See Also nor_SectorRead(), nor_SectorWrite(), nor_BlockReclaim()

Example

```
nor_SectorDiscard(0, 100);
```

int **nor_SectorNum** (uint iID)

Summary Return the total number of sectors on this flash chip.

Details This function returns the total number of sectors to file system.

Pars

nID	The device ID you want to use (the same ID you passed to nor_FlashInit()).

Returns Total number of sectors.

See Also nor_FlashInit(), nor_SectorSize()

Example

```
printf("Total sector number of device %d is %d\n", 0, nor_SectorNum(0));
```

int **nor_SectorSize** (uint iID)

Summary Return the sector size, in bytes, of this flash chip.

Details This function returns the sector size to file system.

Pars

nID	The device ID you want to use (the same ID you passed to nor_FlashInit()).

Returns Flash sector size.

See Also nor_FlashInit(), nor_SectorNum()

Example

```
printf("Sector size of device %d is %d\n", 0, nor_SectorSize(0));
```

int **nor_WearLeveling** (uint iID)

Summary Do static wear leveling.

Details This function checks the wear counter of each block of flash to decide if it is time to do static wear leveling. When the difference between the highest and lowest wear counters is greater than NOR_WEAR_LEVELING_THRESHOLD, this function will move the data in all valid sectors with the block with the lowest wear counter to other blocks and then free this block to be used by later write operations. The application (file system) may call this function when it wants to flush all the cache buffer to the flash memory or during system idle time.

Pars nID The device ID you want to use (the same ID you passed to nor_FlashInit()).

Returns 0

See Also nor_BlockReclaim()

Example

```
nor_WearLeveling(0);
```

5. NOR Flash Hardware IO Routines

The smxNOR flash hardware IO routines are defined in norio.h. These comprise the low-level driver, and they must be implemented for your particular flash hardware and hardware platform. We provide some sample code in the norio.c which can be used as a reference.

It is common for NOR flash to have smaller block sizes at the start or end of the flash (i.e. the boot block). The total size of these will be the same as a normal flash block (e.g. 64KB). To handle this, you can either exclude these small blocks from the part of the flash used by smxNOR or you can treat them as a single block by adding special handling to nor_IO_SectorRead() and nor_IO_SectorWrite().

int **nor_IO_FlashInit** (uint iID, NOR_DEVINFO * pDevInfo);

int **nor_IO_FlashRelease** (uint iID);

int **nor_IO_SectorRead** (uint iID, u8 * pRAMAddr, uint wPhySectorIndex, uint wSectorSize);

int **nor_IO_SectorWrite** (uint iID, u8 * pRAMAddr, uint wPhySectorIndex, uint wSectorSize);

int **nor_IO_BlockErase** (uint iID, u32 wBlockIndex);

int **nor_IO_InfoWrite** (uint iID, void * pInfo, uint wBufSize, u32 wPhyBlockIndex, uint wOffset);

int **nor_IO_InfoRead** (uint iID, void * pInfo, uint wBufSize, u32 wPhyBlockIndex, uint wOffset);

5.1 IO Routine Data Types

```
typedef struct
{
   u32 dwTotalBlockNum;
   u32 dwBlockSize;
} NOR_DEVINFO;
```

5.2 IO Routine Reference

int **nor_IO_FlashInit** (uint iID, NOR_DEVINFO * pDevInfo)

Summary Initialize the flash chip.

Details This function initializes the NOR flash chip and gets the basic information of it.
You need to set the member variables, dwTotalBlockNum and dwBlockSize of structure NOR_DEVINFO, so the driver can initialize its own internal data structures.

Pars

nID	The device ID you want to use.
pDevInfo	The pointer to the device information structure you need to fill.

Returns
1 Initialization succeeded.
0 Initialization failed.

See Also nor_IO_FlashRelease()

Example

```
NOR_DEVINFO DevInfo
If(nor_IO_FlashInit (0, &DevInfo))
{
    printf("Total Block number is %d\n", DevInfo.dwTotalBlockNum);
    printf("Block size is %d\n", DevInfo.dwBlockSize);
}
```

int **nor_IO_FlashRelease** (uint iID)

Summary Release the flash chip.

Details This function releases the NOR flash chip.

Pars nID The device ID you want to use.

Returns 1

See Also nor_IO_FlashInit()

Example

```
nor_IO_FlashRelease (0);
```

int **nor_IO_SectorRead** (uint iID, u8 * pRAMAddr, u32 wPhySectorIndex, uint wSectorSize)

Summary Read one sector of data from the flash chip.

Details This function read one sector of data from the flash chip. The sector is normally 512 bytes by default but may be another size so please make sure the memory buffer is big enough.
If your flash chip cannot read 512 bytes each time, you may need to map this function call to multiple flash commands. For example, some serial flash can only read up to 256 bytes in one command. Our sample code already shows how to handle this case.

Pars

nID	The device ID you want to use.
pRAMAddr	The pointer to the memory buffer to hold the data. The buffer should be at least one sector in size.
wPhySectorIndex	The physical sector index of the flash chip. You may need to map this index to the flash address.
wSectorSize	The size of the buffer.

Returns 1 The read succeeded.

See Also nor_IO_SectorWrite()

Example

```
u8 pData[512];
memset(pData, i, 512);
nor_IO_SectorWrite(0, pData, j + i * iPagePerSec, 512);
memset(pData, 0xFF, 512);
nor_IO_SectorRead(0, pData, j + i * iPagePerSec, 512);
for(k = 0; k < 512; k++)
{
    If(pData[k] != (u8)i)
    {
        printf("IO Sector Write/Read mismatch at %d, %d, %d \r\n", i, j, k);
    }
}
```

int **nor_IO_SectorWrite** (uint iID, u8 * pRAMAddr, u32 wPhySectorIndex, uint wSectorSize)

Summary Write one sector data to the flash chip.

Details This function writes one sector of data to the flash chip. The sector is normally 512 bytes by default. If your flash chip cannot write 512 bytes each time, you may need to map this function call to multiple flash commands. For example, some serial flash can only write up to 256 bytes in one command. Our sample code already shows how to handle this case.

Pars

nID	The device ID you want to use.
pRAMAddr	The pointer to the memory buffer holding the data to write. It should be at least one sector in size.
wPhySectorIndex	The physical sector index of the flash chip. You may need to map this index to the flash address.
wSectorSize	The size of the buffer.

Returns 1 The write succeeded.

See Also nor_IO_SectorRead()

Example

```
u8 pData[512];
memset(pData, i, 512);
nor_IO_SectorWrite(0, pData, j + i * iPagePerSec, 512);
memset(pData, 0xFF, 512);
nor_IO_SectorRead(0, pData, j + i * iPagePerSec, 512);
for(k = 0; k < 512; k++)
{
    If(pData[k] != (u8)i)
    {
        printf("IO Sector Write/Read mismatch at %d, %d, %d \r\n", i, j, k);
    }
}
```

int **nor_IO_InfoRead** (uint iID, void * pInfo, uint wBufSize, u32 wPhyBlockIndex, uint wOffset)

Summary Read a few bytes from the flash chip.

Details This function reads a few bytes of information from the flash chip. Information will only be stored in the first few pages of each block.

Pars

nID	The device ID you want to use.
pInfo	The pointer to the memory buffer pointer holding the data.
wBufSize	The size of the buffer pointed to by pInfo.
wPhyBlockIndex	The physical block index of the flash chip in which the information is stored.
wOffset	The byte offset of the information from the beginning of this block.

Returns 1 The read succeeded.

See Also nor_IO_InfoWrite()

Example

```
nor_IO_InfoWrite(0, &dwInfo, sizeof(u32), i, j*sizeof(u32));
nor_IO_InfoRead(0, &dwInfoTemp, sizeof(u32), i, j*sizeof(u32));
if(dwInfo != dwInfoTemp)
{
      printf("IO Info Write/Read mismatch 1 at %d, %d \r\n", i, j);
}
```

int **nor_IO_InfoWrite** (uint iID, void * pInfo, uint wBufSize, u32 wPhyBlockIndex, uint wOffset)

Summary Write a few bytes to the flash chip.

Details This function writes a few bytes of information to the flash chip. Information will only be stored in the first few pages of each block.

Pars

nID	The device ID you want to use.
pInfo	The pointer to the memory buffer pointer to hold the data.
wBufSize	The size of the buffer pointed to by pInfo.
wPhyBlockIndex	The physical block index of the flash chip in which the information is stored.
wOffset	The byte offset of the information from the beginning of this block.

Returns 1 The write succeeded.

See Also nor_IO_InfoRead()

Example

```
nor_IO_InfoWrite(0, &dwInfo, sizeof(u32), i, j*sizeof(u32));
nor_IO_InfoRead(0, &dwInfoTemp, sizeof(u32), i, j*sizeof(u32));
if(dwInfo != dwInfoTemp)
{
    printf("IO Info Write/Read mismatch 1 at %d, %d \r\n", i, j);
}
```

int **nor_IO_BlockErase** (uint iID, u32 wBlockIndex)

Summary Erase a block of the flash chip.

Details This function erases the whole block at the specified index.

Pars

nID	The device ID you want to use.
wBlockIndex	The physical block index.

Returns 1 Erase succeeded.

See Also nor_IO_SectorRead(), nor_IO_SectorWrite(), nor_IO_InfoRead(), nor_IO_InfoWrite()

Example

```
for(i = 0; i < DevInfo.dwTotalBlockNum; i++)
{
    nor_IO_BlockErase(0, i);
}
```

5.3 Verify the Driver

You need to verify your IO routines before you try any high level APIs such as smxFS and smxFLog.

We provide sample code to verify if your porting of the IO routines is correct. **Please run it first after you complete your porting.** Then test the integration with your file system. There are two ways you can run the verification testing.

1. If you purchased the standalone smxNOR driver without smxFS or smxFlog, you need to copy the code in funciton testNORIO() in XFD**nortest.c** to your application and modify the output message functions (print_err() and print()) to make them work on your system. Run it on your hardware and ensure it does not report any problems. nortest.c also provides other functions to help you verify the smxNOR APIs and measure their performance. testNORFD() is used to test the NOR flash Driver APIs, and testNORIOPerformance() is used to test the IO routines performance.
2. If you also purchased smxFS or smxFlog, we provide APP\DEMO**flltest.c** for low-level flash driver testing. In your project file or makefile, replace fsdemo.c, fstest.c, ffsdemo.c, fldemo.c, or fltest.c with flltest.c, and re-build the application. The low-level driver testing code will run instead of the normal smxFS/FFS/FLog demo code.

Even you are running the driver verified by Micro Digital, we still recommend you run the low level testing code to make sure it is working properly on your hardware.

5.4 Erase Your Flash First

smxNOR assumes your NOR flash is empty. If you have some preloaded data in the flash or run some 3rd party test code on it that does not erase the flash after it is done, you may need to erase the flash first, before you run smxNOR. To do that:

1. Run the IO routine testing in flltest.c, as indicated in section 5.3 Verify the Driver. The last step of that testing will erase the whole flash again.
2. If you are sure your IO routines are working, set ERASE_FLASH_ONLY to 1 at the top of flltest.c, which will cause the test code to only erase the whole flash. This will save you some time.

6. Application Notes

6.1 Support Multiple Flash Chips as One Big Disk

If you want to use multiple flash chips to get one big size flash disk, such as two 32MB flash to get one 64 MB flash disk. You need to do some thing special in the low level driver norio_xxx.c.

a. In nor_IO_FlashInit(), you need to set pDevInfo->dwTotalBlockNum as the total block number of all the flash chip you have. For example, if the 32MB flash has 256 128KB blocks, set pDevInfo->dwTotalBlockNum to 2*256.

b. In nor_IO_SectorRead/Write(), nor_IO_InfoRead/Write() and nor_IO_BlockErase(), You need to check the parameter wSectorIndex or wBlockIndex to decide which flash chip you need to use. For example,

```
int  nor_IO_SectorRead (uint iID, u8 * pRAMAddr, uint wSectorIndex, uint wSectorSize)
{
   if(wSectorIndex < 256*128)
   {
   /* use flash chip 0 */
   }
   else
   {
   /* use flash chip 1 */
   }
}
int  nor_IO_BlockErase (uint iID, u32 wBlockIndex)
{
   if(wBlockIndex < 256)
   {
   /* use flash chip 0 */
   }
   else
   {
   /* use flash chip 1 */
   }
}
```

6.2 Support Multiple Flash Chips as Multiple Disks

If you want to use multiple flash chips to get multiple flash disks, such as two 32MB flash, chip 0 is disk A: and chip 1 is disk B: You need to do some thing special in the low level driver norio_xxx.c.

a. Set NOR_MAX_CHIP_NUM to the chip number you want in fdcfg.h

b. In nor_IO_SectorRead/Write(), nor_IO_InfoRead/Write() and nor_IO_BlockErase(), You need to check the parameter iID to decide which flash chip you need to use. For example,

```
int  nor_IO_SectorRead (uint iID, u8 * pRAMAddr, uint wSectorIndex, uint wSectorSize)
{
   u32 dwBaseAddr;
```

```
    if(iID == 0)
    {
       dwBaseAddr = NOR_FLASH_BASE_ADDR0;
    }
    else
    {
       dwBaseAddr = NOR_FLASH_BASE_ADDR1;
    }
     memcpy(pRAMAddr, (u8 *)(dwBaseAddr + wSectorIndex*wSectorSize), wSectorSize);
    return 1;
}
```

If these two flash need different driver, You may need to implement and test those two driver first and then use a separate shell to interface the low level driver with the smxNOR flash driver. For example, you need to use the processor's internal flash as disk A: for some configuration file and external flash chip as B: for data file.
You may implement the internal flash driver as
nor_IO_Internal_FlashInit()
nor_IO_Internal_SectorRead()
nor_IO_Internal_SectorWrite()
nor_IO_Internal_InfoRead()
nor_IO_Internal_InfoWrite()
nor_IO_Internal_ BlockErase ()
You may implement the external flash driver as
nor_IO_External_FlashInit()
nor_IO_External_SectorRead()
nor_IO_External_SectorWrite()
nor_IO_External_InfoRead()
nor_IO_External_InfoWrite()
nor_IO_External_ BlockErase ()

finally your low level driver should be something like

```
int  nor_IO_FlashInit (uint iID, NOR_DEVINFO * pDevInfo)
{
    if(iID == 0)
    {
       return nor_IO_Internal_FlashInit(pDevInfo);
    }
    else
    {
       return nor_IO_External_FlashInit(pDevInfo);
    }
}

int  nor_IO_SectorRead (uint iID, u8 * pRAMAddr, uint wSectorIndex, uint wSectorSize){
    if(iID == 0)
    {
       return nor_IO_Internal_ SectorRead (pRAMAddr, wSectorIndex, wSectorSize);
    }
    else
    {
```

```
        return nor_IO_External_ SectorRead (pRAMAddr, wSectorIndex, wSectorSize);
    }
}

int  nor_IO_SectorWrite (uint iID, u8 * pRAMAddr, uint wSectorIndex, uint wSectorSize){
    if(iID == 0)
    {
        return nor_IO_Internal_ SectorWrite (pRAMAddr, wSectorIndex, wSectorSize);
    }
    else
    {
        return nor_IO_External_ SectorWrite (pRAMAddr, wSectorIndex, wSectorSize);
    }
}

int  nor_IO_InfoRead (uint iID, void * pInfo, uint wBufSize, u32 wBlockIndex, uint wOffset)
{
    if(iID == 0)
    {
        return nor_IO_Internal_ InfoRead (pInfo, wBufSize, wBlockIndex, wOffset);
    }
    else
    {
        return nor_IO_External_ InfoRead (pInfo, wBufSize, wBlockIndex, wOffset);
    }
}

int  nor_IO_InfoWrite (uint iID, void * pInfo, uint wBufSize, u32 wBlockIndex, uint wOffset)
{
    if(iID == 0)
    {
        return nor_IO_Internal_ InfoWrite (pInfo, wBufSize, wBlockIndex, wOffset);
    }
    else
    {
        return nor_IO_External_ InfoWrite (pInfo, wBufSize, wBlockIndex, wOffset);
    }
}

int  nor_IO_BlockErase (uint iID, u32 wBlockIndex)
{
    if(iID == 0)
    {
        return nor_IO_Internal_ BlockErase (wBlockIndex);
    }
    else
    {
        return nor_IO_External_ BlockErase (wBlockIndex);
    }
}
```

6.3 Porting for CFI-Compatible Flash

We provide a general-purpose CFI-compatible NOR flash driver, norio_cfi.c. It should handle most common features of CFI flash but you still need to do some porting which is unique to your hardware. Basically you need to implement the following macros:

NOR_FLASH_BASE is the base address of your flash chip, such as 0x80000000. You need to check your hardware design and processor's manual to get the correct setting for this.

PROGRAM_TIMEOUT is the timeout (status read loops) to program one word of data. It depends on your flash chip and CPU speed. You may need to use a big value first and then reduce it.

ERASE_TIMEOUT is the timeout (status read loops) to erase one block of data. It depends on your flash chip and CPU speed. Erasing a block is much slower than programming a word of data. You may need to use a big value first and then reduce it.

PROGRAM_RETRY is the number of times to retry if a word program attempt fails. The default value is probably fine.

HARDWARE_INIT initializes the NOR flash bus. You may need to set up the pin multiplexing and the proper timing and width of this bus, and enable the clock and power to the external memory controller, etc. Normally the chip vendor may provide some sample code for this, which you can use. See our implementations for other hardware platforms.

DEV_TYPE is the flash chip data width. It is defined as the data type. Most NOR flash are 16-bit devices so use u16.

BUS_TYPE is the external memory bus width to the NOR flash chip. Most likely it should be u16 but it is possible to use 8-bit bus to connect to a 16-bit flash chip. For that case, use u8.

A. File Summary

FILE	DESCRIPTION
fdcfg.h	Configuration file for smxNOR.
fport.h	Porting definitions, macros, and functions for hardware and OS. Based on smxBase.
norfd.c, .h	NOR flash driver API.
norio*.c, .h	NOR flash driver hardware IO routines.
nortest.c	Sample code to verify API and hardware IO routines

B. Porting Notes

The porting layer is simple. It does not need any OS system calls. The most important part of smxNOR porting is to implement the NOR flash hardware IO routines, according to your specific hardware. We already provide the sample code for M25PXX serial flash driver in norio.c. You can use it as a starting point. Please read section 4 NOR Flash Hardware IO Routines for details about how to implement those functions.

B.1 C Library Function Requirements

This is a list of C library functions that smxNOR calls. If your compiler does not provide some of these, you should implement them yourself.

- memcpy()
- memcmp()
- memset()

C. Tested Hardware

- STMicro M25P10, M25P80 on STR710-Eval board
- STMicro M25P16 on our Avnet Coldfire 5282 add-on board.
- 39VF320 NOR Flash on LPC2468 board
- 28F128K3, 28F256K3, 28F128J3D NOR Flash on MCF5485EVB board

D. Preprogramming Flash

If you solder a new flash chip to your board and run your application that includes smxNOR plus smxFS or other filesystem, the filesystem structure will be created on the flash chip automatically. The software takes care of the details of doing the low-level flash format (including marking any bad blocks encountered), and formatting it with the high-level filesystem (e.g. smxFFS or FAT). If your device must have some files already saved in the filesystem, one approach is to run your device and copy the files to it. But this process may be too time consuming.

For manufacturing, it is convenient to be able to preprogram the flash chips before soldering them to the board, especially to program several at once (gang programming). However, this is complicated because each flash chip may have bad blocks in different locations, so the image that must be written to each may vary. With some flash programmers it is possible to define an algorithm for programming the flash, but this is complex and problematic because:

1. The algorithm must be changed if any changes are made to the internals of the flash driver (smxNOR).

2. The algorithm differs depending on the high-level filesystem (FAT12, FAT16, FAT32).

3. It is not possible to gang-program the devices at the same time because the bad blocks are in different places. If a gang programmer is used, all flash chips must be programmed individually.

Our solution is a hybrid of the two approaches. It makes the assumption that the first n flash blocks on a device are almost always good for the first few cycles of writing, where n is the number of flash blocks needed to store the initial image. Typically, the amount of space occupied by the initial files is a small fraction of total disk space. These are the steps we recommend:

1. Use our **FlashImage** utility to create an image of your flash (BIN\FlashImage). This utility creates the image in a file on your development PC. This image assumes there are no bad blocks in this area of the flash. A config file (cfg.h) is used to specify the flash type and list the files to store in the image. It is necessary to configure and build this utility. See the readme in its directory for directions.

2. Supply this image to your gang programmer to program all devices simultaneously.

3. Do a verify operation on each chip. The ones that pass are soldered to the boards. The others are collected; they can be programmed manually by running the application software on them. We expect that a very small percentage will require this.

Any bad blocks in the remainder of the media will be handled as encountered during normal use of your device.

The key point is that the utility is built using the same flash driver and filesystem code that is linked to your application. If any changes are made to the internals of the flash driver, it is only necessary to recompile the utility. It is not necessary to create and maintain complicated flash programmer files.

Note: The above solution cannot be used when you need to preprogram a large amount of data in the flash chip, because the bigger the image is, the more likely it is to span an area that has bad blocks. The smaller your image is, the higher your preprogramming yields will be.